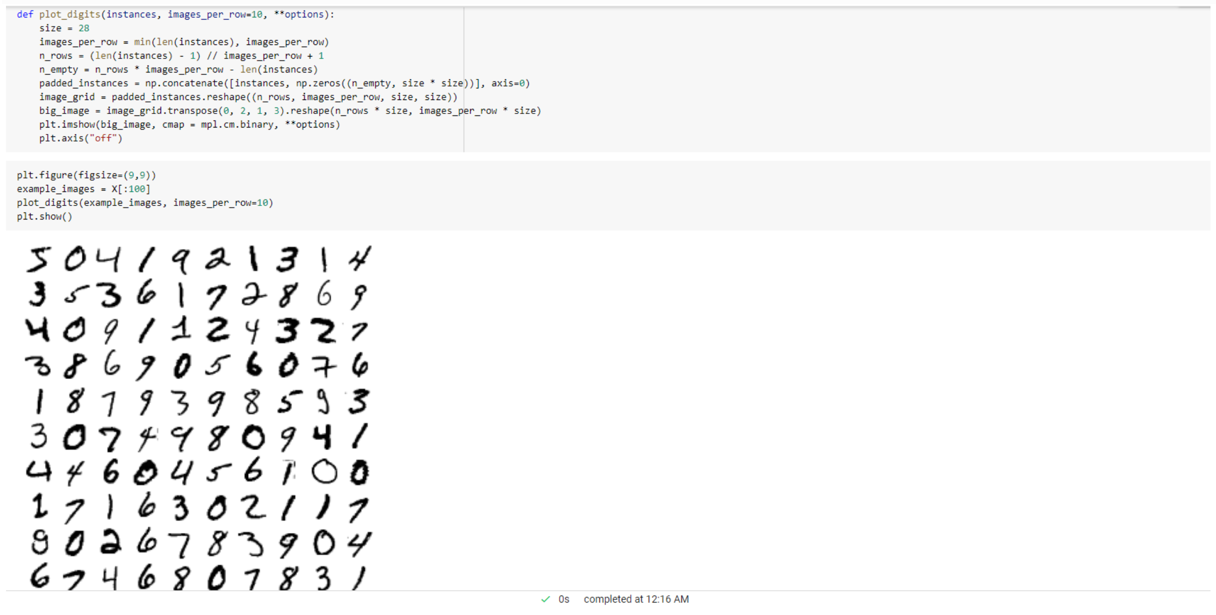
Analysis of different Classification Algorithms on MNIST Datasets:

About MNIST:
The MNIST database of handwritten digits with 784 features. It is a portion of a larger set that is made public by NIST. The digits have been centered in a fixed-size image and size-normalized. To fit in a 20x20 pixel box while maintaining their aspect ratio, the original black and white (bilevel) photographs from NIST were size normalized. As a result of the normalization algorithm's anti-aliasing approach, the final photos have grey levels. By calculating the center of mass of the pixels and translating the image to place this point in the middle of the 28x28 field, the images were centered in a 28x28 image.

We started by *Reorganizing the array as a grid with 28*28 images.and plotting.*



```python
def plot_digits(instances, images_per_row=10, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    n_rows = (len(instances) - 1) // images_per_row + 1
    n_empty = n_rows * images_per_row - len(instances)
    padded_instances = np.concatenate([instances, np.zeros((n_empty, size * size))], axis=0)
    image_grid = padded_instances.reshape((n_rows, images_per_row, size, size))
    big_image = image_grid.transpose(0, 2, 1, 3).reshape(n_rows * size, images_per_row * size)
    plt.imshow(big_image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

```python
plt.figure(figsize=(9,9))
example_images = X[:100]
plot_digits(example_images, images_per_row=10)
plt.show()
```

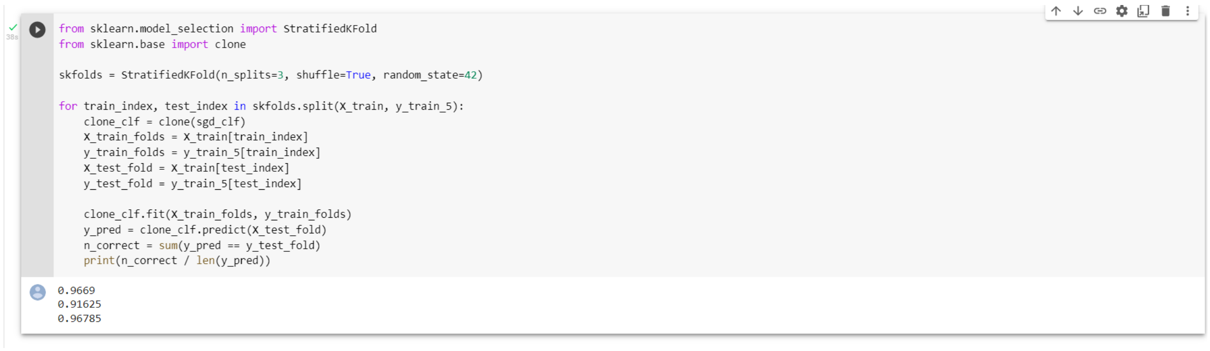
```
✓ 0s    completed at 12:16 AM
```

1st Classifier - Binary Classifier

Training and testing for Class 5: True for all 5 and false for others. Have used SGDClassifier

Performance Measures:

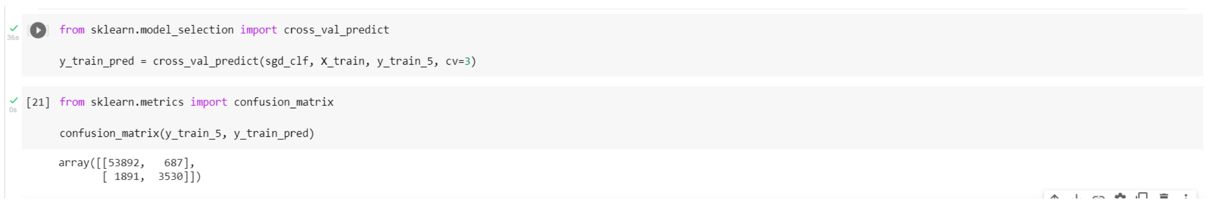
1. Measuring Accuracy Using Cross-Validation



```python
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred))
```

```
0.9669
0.91625
0.96785
```

2. Confusion Matrix

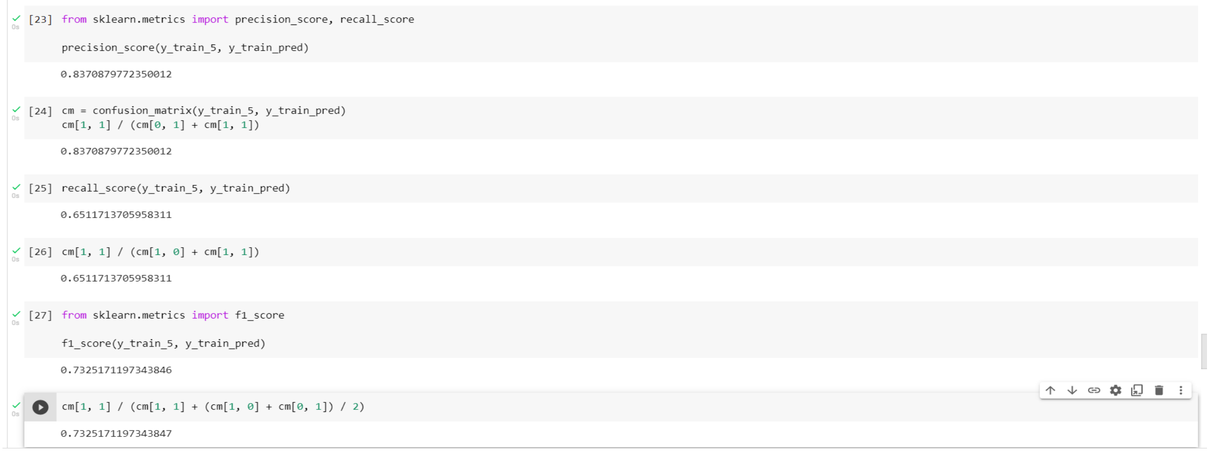


```python
from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

```python
[21] from sklearn.metrics import confusion_matrix

confusion_matrix(y_train_5, y_train_pred)
```

```
array([[53892,   687],
       [ 1891,  3530]])
```

3. Precision and Recall



```python
[23] from sklearn.metrics import precision_score, recall_score

precision_score(y_train_5, y_train_pred)
```

```
0.8370879772350012
```

```python
[24] cm = confusion_matrix(y_train_5, y_train_pred)
cm[1, 1] / (cm[0, 1] + cm[1, 1])
```

```
0.8370879772350012
```

```python
[25] recall_score(y_train_5, y_train_pred)
```

```
0.6511713705958311
```

```python
[26] cm[1, 1] / (cm[1, 0] + cm[1, 1])
```

```
0.6511713705958311
```

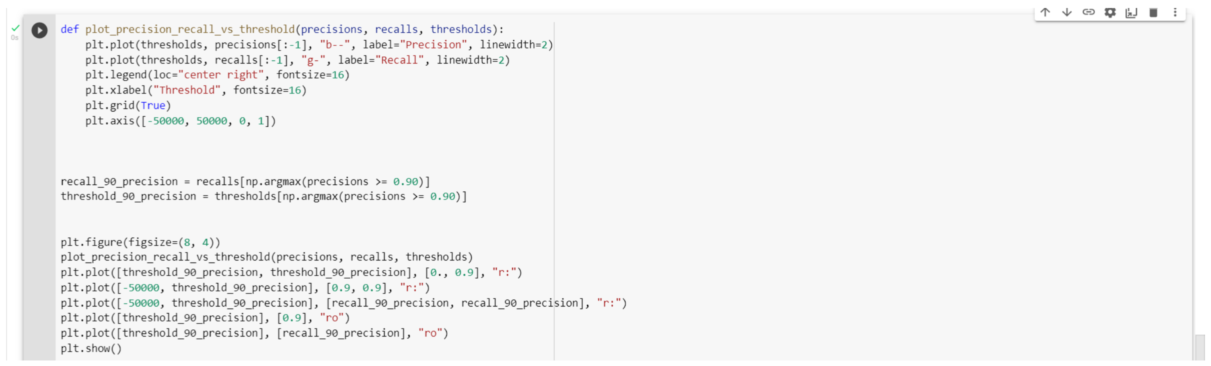
```python
[27] from sklearn.metrics import f1_score

f1_score(y_train_5, y_train_pred)
```

```
0.7325171197343846
```

```python
cm[1, 1] / (cm[1, 1] + (cm[1, 0] + cm[0, 1]) / 2)
```

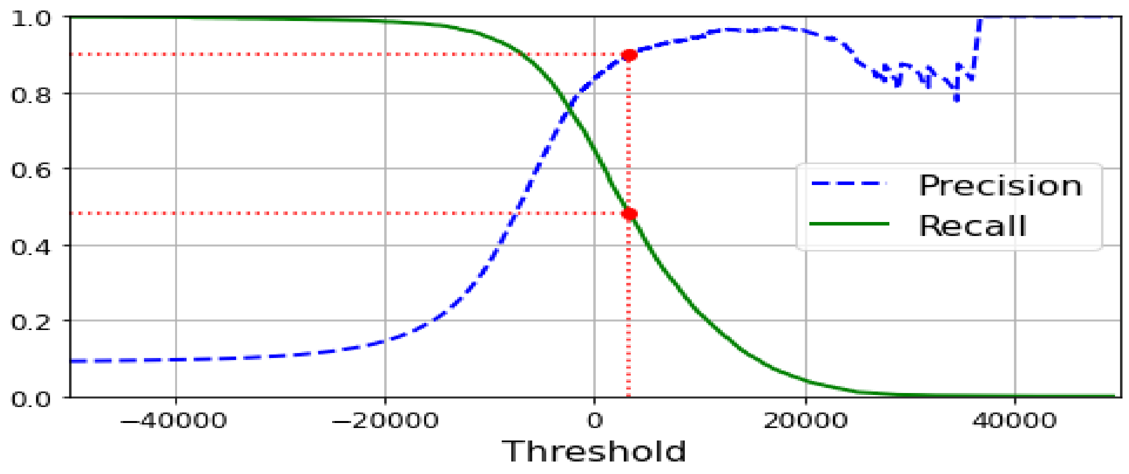
```
0.7325171197343847
```

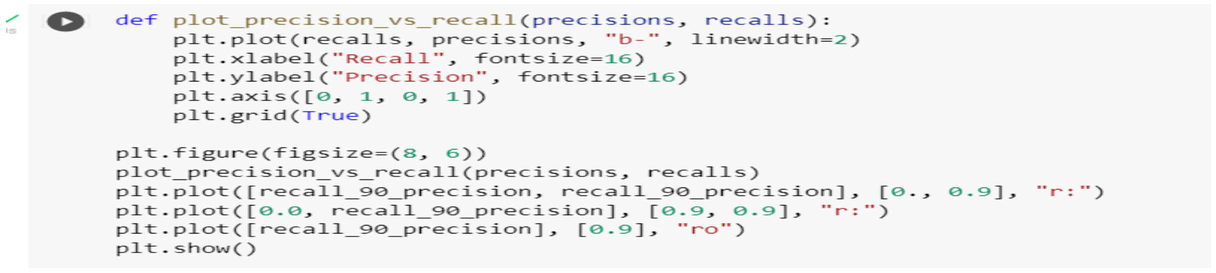
4. Precision/Recall Trade-off



```python
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.legend(loc="center right", fontsize=16)
    plt.xlabel("Threshold", fontsize=16)
    plt.grid(True)
    plt.axis([-50000, 50000, 0, 1])



recall_90_precision = recalls[np.argmax(precisions >= 0.90)]
threshold_90_precision = thresholds[np.argmax(precisions >= 0.90)]


plt.figure(figsize=(8, 4))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.plot([threshold_90_precision, threshold_90_precision], [0., 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [0.9, 0.9], "r:")
plt.plot([-50000, threshold_90_precision], [recall_90_precision, recall_90_precision], "r:")
plt.plot([threshold_90_precision], [0.9], "ro")
plt.plot([threshold_90_precision], [recall_90_precision], "ro")
plt.show()
```

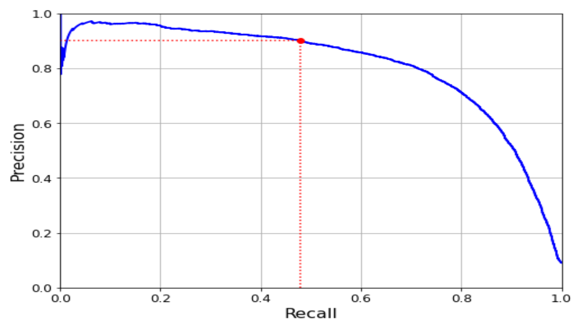
precision, recall_vs_threshold_plot

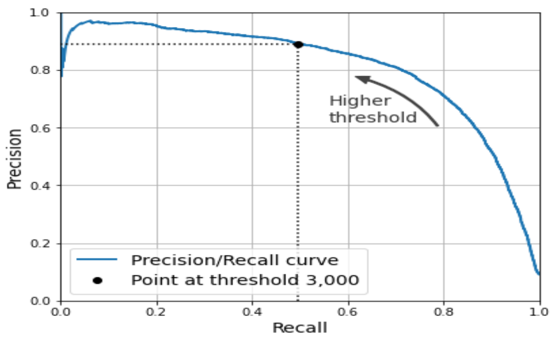




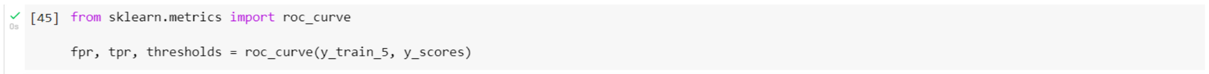
```
def plot_precision_vs_recall(precisions, recalls):
    plt.plot(recalls, precisions, "b-", linewidth=2)
    plt.xlabel("Recall", fontsize=16)
    plt.ylabel("Precision", fontsize=16)
    plt.axis([0, 1, 0, 1])
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_precision_vs_recall(precisions, recalls)
plt.plot([recall_90_precision, recall_90_precision], [0., 0.9], "r:")
plt.plot([0.0, recall_90_precision], [0.9, 0.9], "r:")
plt.plot([recall_90_precision], [0.9], "ro")
plt.show()
```

precision_vs_recall_plot

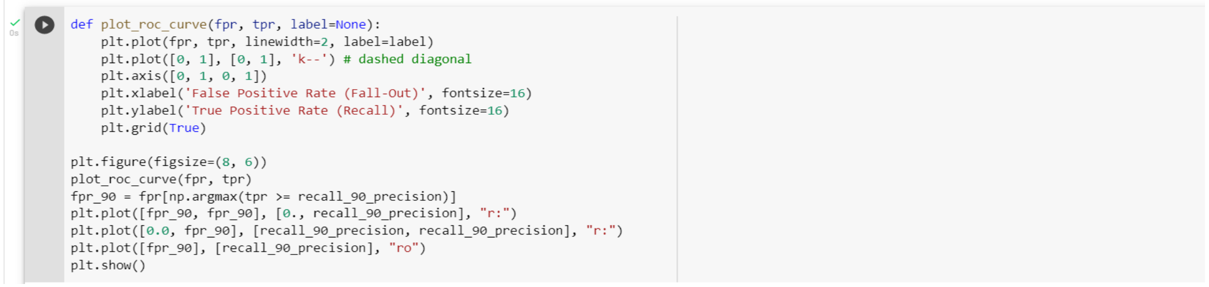




5. ROC Curve

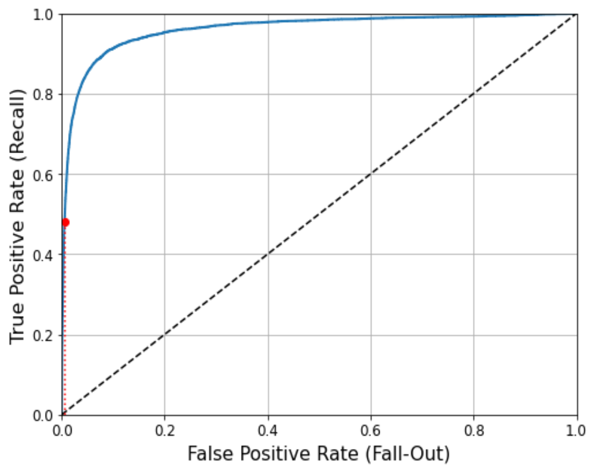


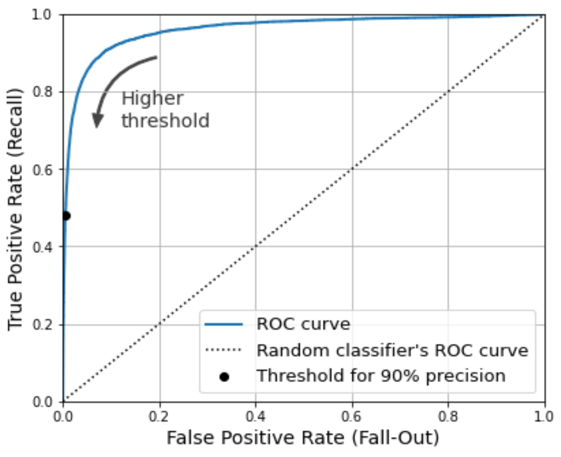
```
[45] from sklearn.metrics import roc_curve

     fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```



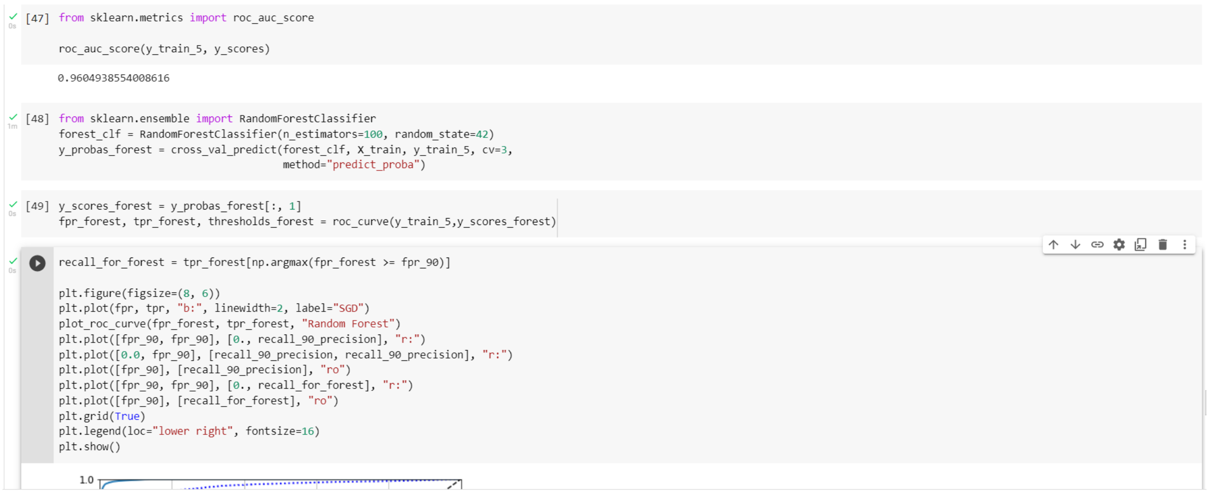
```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # dashed diagonal
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (Fall-Out)', fontsize=16)
    plt.ylabel('True Positive Rate (Recall)', fontsize=16)
    plt.grid(True)

plt.figure(figsize=(8, 6))
plot_roc_curve(fpr, tpr)
fpr_90 = fpr[np.argmax(tpr >= recall_90_precision)]
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
plt.plot([fpr_90], [recall_90_precision], "ro")
plt.show()
```

Roc_curve_plot for SGD





To improve the performance of the binary classification we have ensemble Random Forest algorithm.



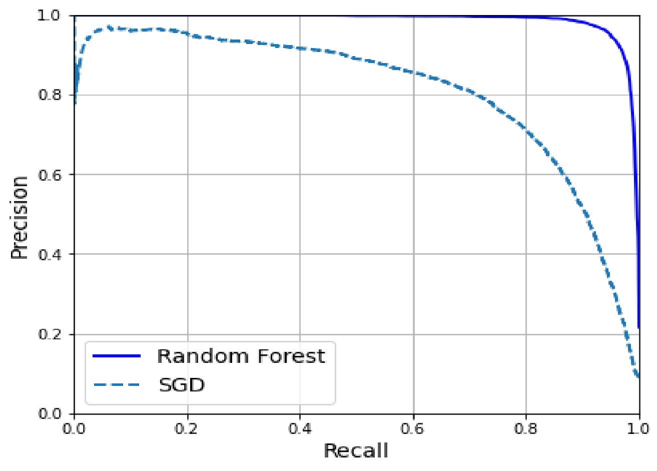
```
[47] from sklearn.metrics import roc_auc_score

     roc_auc_score(y_train_5, y_scores)

     0.9604938554008616
```

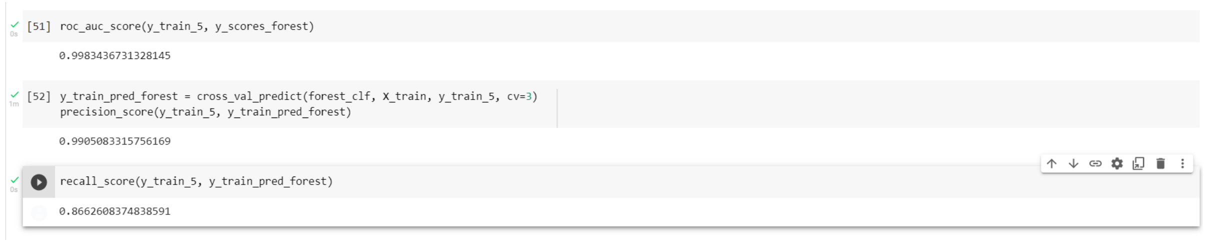
```
[48] from sklearn.ensemble import RandomForestClassifier
     forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
     y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                         method="predict_proba")
```

```
[49] y_scores_forest = y_probas_forest[:, 1]
     fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

```
recall_for_forest = tpr_forest[np.argmax(fpr_forest >= fpr_90)]

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.plot([fpr_90, fpr_90], [0., recall_90_precision], "r:")
plt.plot([0.0, fpr_90], [recall_90_precision, recall_90_precision], "r:")
plt.plot([fpr_90], [recall_90_precision], "ro")
plt.plot([fpr_90, fpr_90], [0., recall_for_forest], "r:")
plt.plot([fpr_90], [recall_for_forest], "ro")
plt.grid(True)
plt.legend(loc="lower right", fontsize=16)
plt.show()
```

pr_curve_comparison_plot





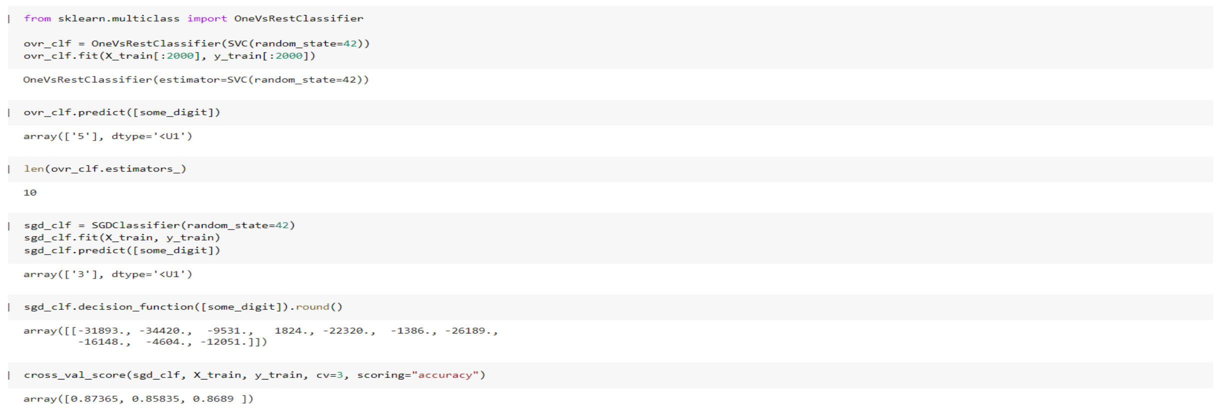
```
[51] roc_auc_score(y_train_5, y_scores_forest)

     0.9983436731328145

[52] y_train_pred_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3)
     precision_score(y_train_5, y_train_pred_forest)

     0.9905083315756169

     recall_score(y_train_5, y_train_pred_forest)

     0.8662608374838591
```

Classification Algorithm Name	**AUC**	**Precision**	**Recall**
Binary Classification using SGD classifier	0.964	0.837	0.651
Ensemble using Random Forest	0.998	0.990	0.886

From the table above we can clearly see by ensembling random forest with binary classifier the overall performance gets improved.

2nd Classifier- Multiclass Classification

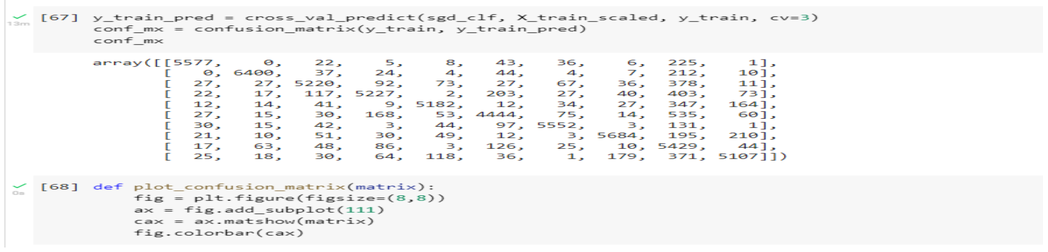
Performance Evaluation:



```
from sklearn.multiclass import OneVsRestClassifier

ovr_clf = OneVsRestClassifier(SVC(random_state=42))
ovr_clf.fit(X_train[:2000], y_train[:2000])

OneVsRestClassifier(estimator=SVC(random_state=42))

ovr_clf.predict([some_digit])

array(['5'], dtype='<U1')

len(ovr_clf.estimators_)

10

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])

array(['3'], dtype='<U1')

sgd_clf.decision_function([some_digit]).round()

array([[-31893., -34420.,  -9531.,   1824., -22320.,  -1386., -26189.,
        -16148.,  -4604., -12051.]])

cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")

array([0.87365, 0.85835, 0.8689 ])
```

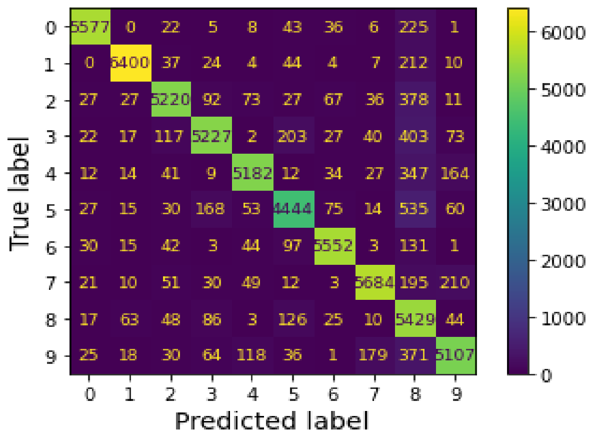
Accuracy: 86.89%: As the Accuracy is very low we have done error analysis below.

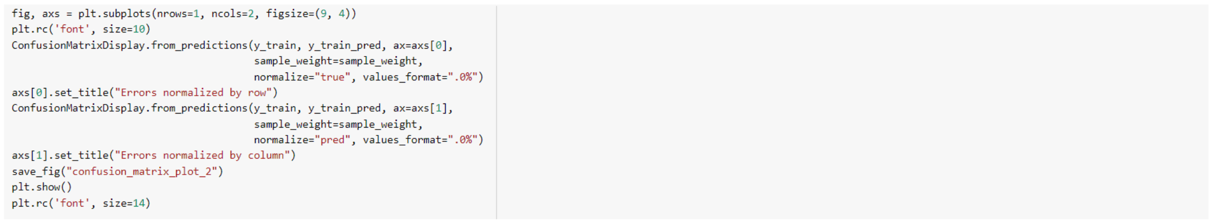
Error Analysis (this is done so as to understand how to improve the accuracy)



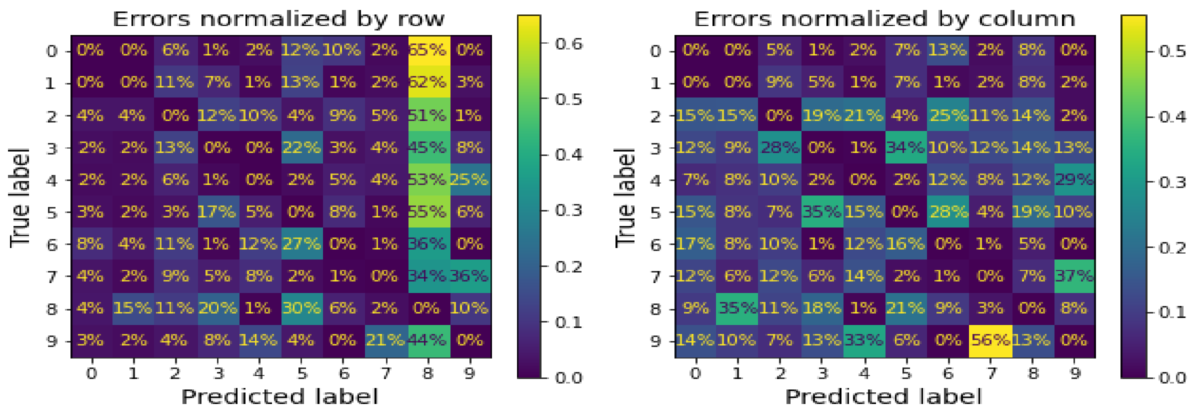
```
[67] y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
     conf_mx = confusion_matrix(y_train, y_train_pred)
     conf_mx

     array([[5577,    0,   22,    5,    8,   43,   36,    6,  225,    1],
            [   0, 6400,   37,   24,    4,   44,    4,    7,  212,   10],
            [  27,   27, 5220,   92,   73,   27,   67,   36,  378,   11],
            [  22,   17,  117, 5227,    2,  203,   27,   40,  403,   73],
            [  12,   14,   41,    9, 5182,   12,   34,   27,  347,  164],
            [  27,   15,   30,  168,   53, 4444,   75,   14,  535,   60],
            [  30,   15,   42,    3,   44,   97, 5552,    3,  131,    1],
            [  21,   10,   51,   30,   49,   12,    3, 5684,  195,  210],
            [  17,   63,   48,   86,    3,  126,   25,   10, 5429,   44],
            [  25,   18,   30,   64,  118,   36,    1,  179,  371, 5107]])

[68] def plot_confusion_matrix(matrix):
         fig = plt.figure(figsize=(8,8))
         ax = fig.add_subplot(111)
         cax = ax.matshow(matrix)
         fig.colorbar(cax)
```

Confusion Matrix



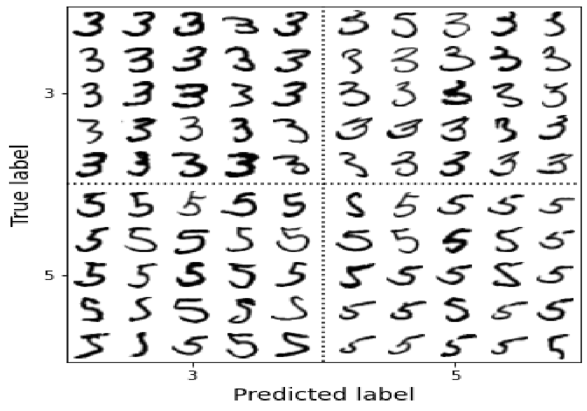


```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(9, 4))
plt.rc('font', size=10)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, ax=axs[0],
                                        sample_weight=sample_weight,
                                        normalize="true", values_format=".0%")
axs[0].set_title("Errors normalized by row")
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred, ax=axs[1],
                                        sample_weight=sample_weight,
                                        normalize="pred", values_format=".0%")
axs[1].set_title("Errors normalized by column")
save_fig("confusion_matrix_plot_2")
plt.show()
plt.rc('font', size=14)
```



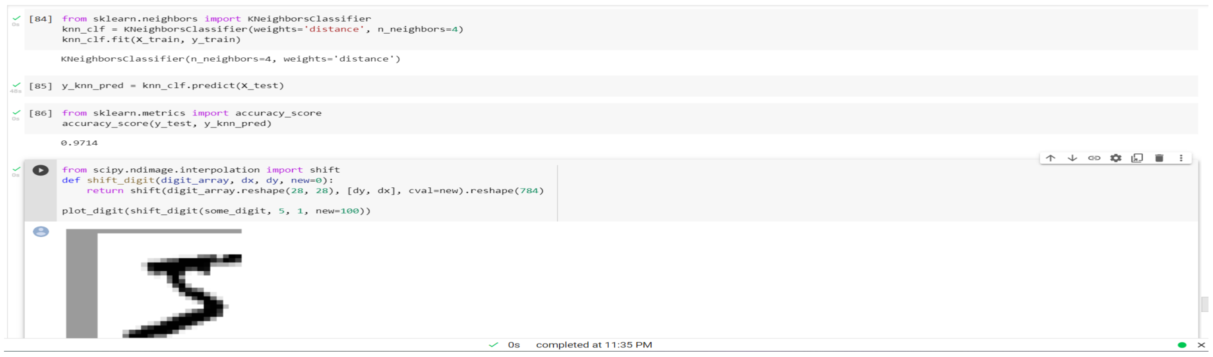
```
cl_a, cl_b = '3', '5'
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]
```

```
size = 5
pad = 0.2
plt.figure(figsize=(size, size))
for images, (label_col, label_row) in [(X_ba, (0, 0)), (X_bb, (1, 0)),
                                        (X_aa, (0, 1)), (X_ab, (1, 1))]:
    for idx, image_data in enumerate(images[:size*size]):
        x = idx % size + label_col * (size + pad)
        y = idx // size + label_row * (size + pad)
        plt.imshow(image_data.reshape(28, 28), cmap="binary",
                   extent=(x, x + 1, y, y + 1))
plt.xticks([size / 2, size + pad + size / 2], [str(cl_a), str(cl_b)])
plt.yticks([size / 2, size + pad + size / 2], [str(cl_b), str(cl_a)])
plt.plot([size + pad / 2, size + pad / 2], [0, 2 * size + pad], "k:")
plt.plot([0, 2 * size + pad], [size + pad / 2, size + pad / 2], "k:")
plt.axis([0, 2 * size + pad, 0, 2 * size + pad])
plt.xlabel("Predicted label")
plt.ylabel("True label")
save_fig("error_analysis_digits_plot")
plt.show()
```

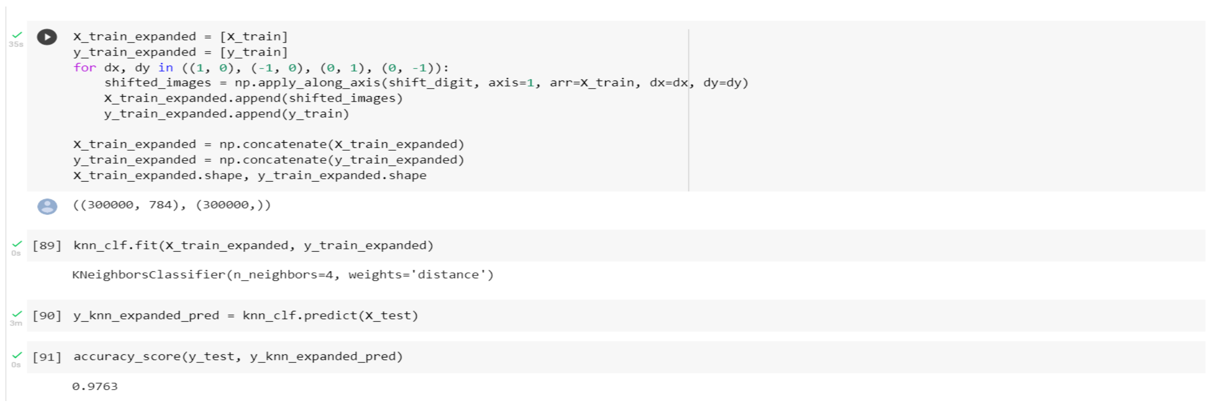


3rd Classifier- KNN classifier

Performance Evaluation:



```
[84] from sklearn.neighbors import KNeighborsClassifier
     knn_clf = KNeighborsClassifier(weights='distance', n_neighbors=4)
     knn_clf.fit(X_train, y_train)

     KNeighborsClassifier(n_neighbors=4, weights='distance')

[85] y_knn_pred = knn_clf.predict(X_test)

[86] from sklearn.metrics import accuracy_score
     accuracy_score(y_test, y_knn_pred)

     0.9714

     from scipy.ndimage.interpolation import shift
     def shift_digit(digit_array, dx, dy, new=0):
         return shift(digit_array.reshape(28, 28), [dy, dx], cval=new).reshape(784)

     plot_digit(shift_digit(some_digit, 5, 1, new=100))
```



```
X_train_expanded = [X_train]
y_train_expanded = [y_train]
for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
    shifted_images = np.apply_along_axis(shift_digit, axis=1, arr=X_train, dx=dx, dy=dy)
    X_train_expanded.append(shifted_images)
    y_train_expanded.append(y_train)

X_train_expanded = np.concatenate(X_train_expanded)
y_train_expanded = np.concatenate(y_train_expanded)
X_train_expanded.shape, y_train_expanded.shape
```

```
((300000, 784), (300000,))
```

```
[89] knn_clf.fit(X_train_expanded, y_train_expanded)
```

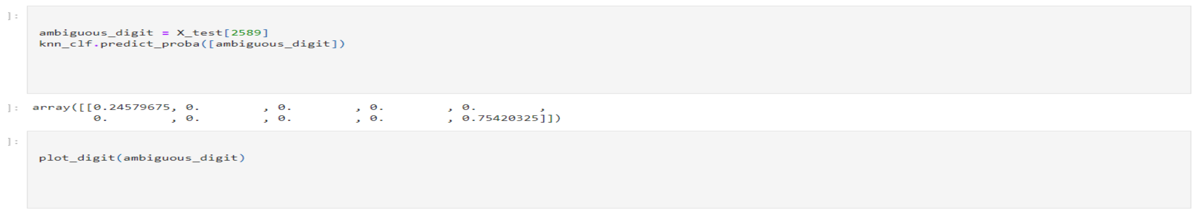
```
KNeighborsClassifier(n_neighbors=4, weights='distance')
```

```
[90] y_knn_expanded_pred = knn_clf.predict(X_test)
```

```
[91] accuracy_score(y_test, y_knn_expanded_pred)
```

```
0.9763
```

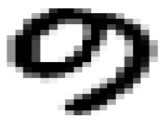
Accuracy : 97.63%



```
]:  ambiguous_digit = X_test[2589]
    knn_clf.predict_proba([ambiguous_digit])
```

```
]:  array([[0.24579675, 0.        , 0.        , 0.        , 0.        ,
            0.        , 0.        , 0.        , 0.        , 0.75420325]])
```

```
]:  plot_digit(ambiguous_digit)
```



Here we see for the image : knn_classifier_predicted this image could have possibilities of being a digit in the array format starting from 0 - 9 :

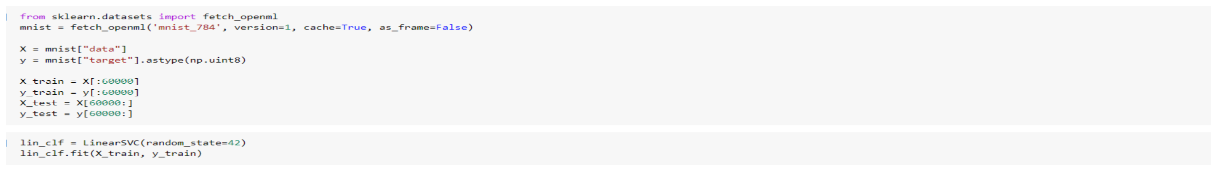
[0.24579675, 0., 0., 0., 0., 0., 0., 0., 0., 0.75420325]: which resembles about 24.57% of digit 0 and 75.42& of digit 9:

4th classifier- Linear SVM classifier

As SVM are binary classifiers, we must use one-vs-rest to classify all ten digits. To speed up the process, we may want to tune the hyperparameters using small validation sets.

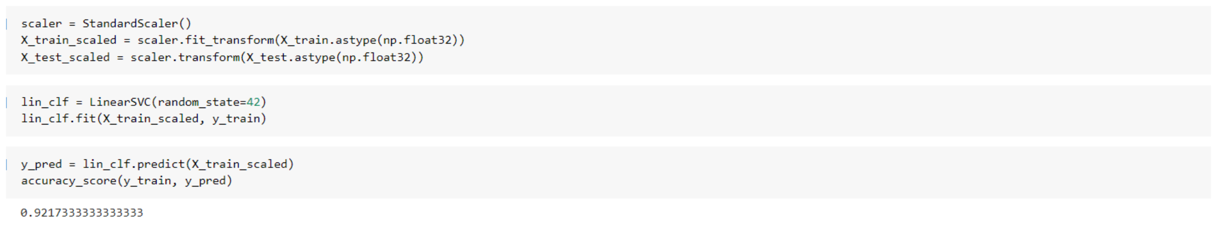
We have taken 60,000 instances to train the data and last 10,000 to test.

Performance Evaluation:



```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, cache=True, as_frame=False)

X = mnist["data"]
y = mnist["target"].astype(np.uint8)

X_train = X[:60000]
y_train = y[:60000]
X_test = X[60000:]
y_test = y[60000:]
```

```
lin_clf = LinearSVC(random_state=42)
lin_clf.fit(X_train, y_train)
```



```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train.astype(np.float32))
X_test_scaled = scaler.transform(X_test.astype(np.float32))
```

```
lin_clf = LinearSVC(random_state=42)
lin_clf.fit(X_train_scaled, y_train)
```

```
y_pred = lin_clf.predict(X_train_scaled)
accuracy_score(y_train, y_pred)
```

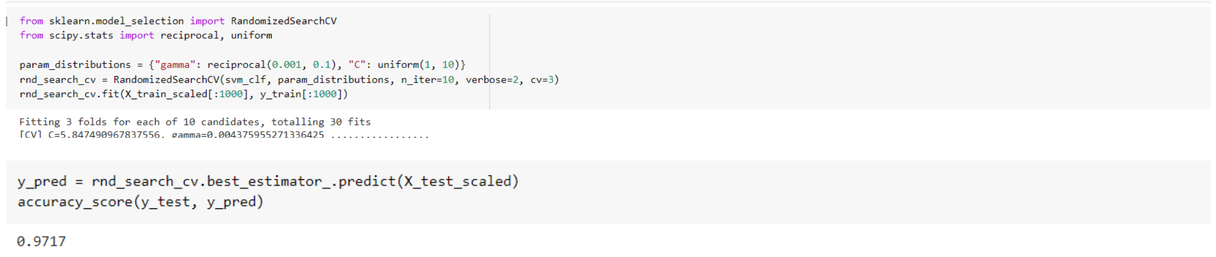
```
0.9217333333333333
```

Accuracy using Linear SVM we got **92.17%.**

5th Classifier - SVM with RBF Kernel

We Tune the hyperparameters by doing a randomized search with cross validation for 1000 instances.

Performance Analysis:



```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import reciprocal, uniform

param_distributions = {"gamma": reciprocal(0.001, 0.1), "C": uniform(1, 10)}
rnd_search_cv = RandomizedSearchCV(svm_clf, param_distributions, n_iter=10, verbose=2, cv=3)
rnd_search_cv.fit(X_train_scaled[:1000], y_train[:1000])
```

```
Fitting 3 folds for each of 10 candidates, totalling 30 fits
[CV] C=5.847490967837556, gamma=0.004375955271336425 .................
```

```
y_pred = rnd_search_cv.best_estimator_.predict(X_test_scaled)
accuracy_score(y_test, y_pred)
```

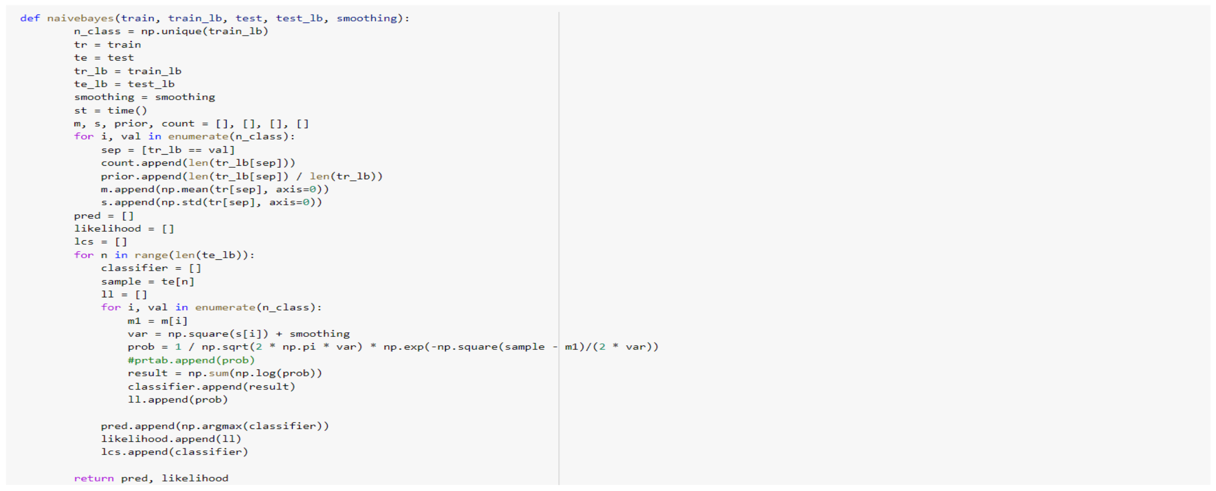
```
0.9717
```

Accuracy —> 97.17%

Some people in ML platforms got the accuracy about 98% by using different hyperparameters values.(with C of 5 and gamma was around 0.004-0.006, it was taking time so haven't tune on this data)

6th classifier- Naive Bayes classifier

Performance Calculation on MNIST dataset:



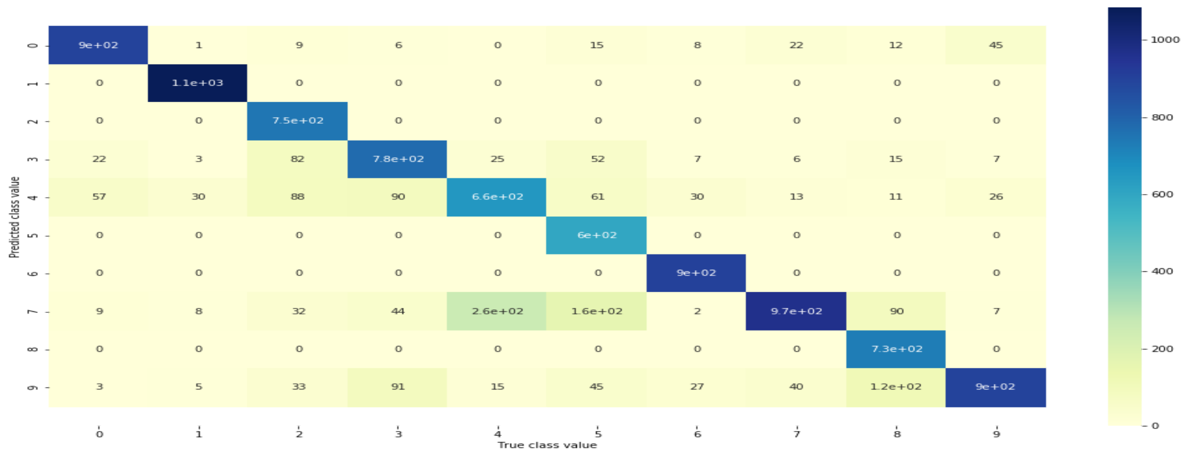
```
def naivebayes(train, train_lb, test, test_lb, smoothing):
        n_class = np.unique(train_lb)
        tr = train
        te = test
        tr_lb = train_lb
        te_lb = test_lb
        smoothing = smoothing
        st = time()
        m, s, prior, count = [], [], [], []
        for i, val in enumerate(n_class):
            sep = [tr_lb == val]
            count.append(len(tr_lb[sep]))
            prior.append(len(tr_lb[sep]) / len(tr_lb))
            m.append(np.mean(tr[sep], axis=0))
            s.append(np.std(tr[sep], axis=0))
        pred = []
        likelihood = []
        lcs = []
        for n in range(len(te_lb)):
            classifier = []
            sample = te[n]
            ll = []
            for i, val in enumerate(n_class):
                m1 = m[i]
                var = np.square(s[i]) + smoothing
                prob = 1 / np.sqrt(2 * np.pi * var) * np.exp(-np.square(sample - m1)/(2 * var))
                #prtab.append(prob)
                result = np.sum(np.log(prob))
                classifier.append(result)
                ll.append(prob)

            pred.append(np.argmax(classifier))
            likelihood.append(ll)
            lcs.append(classifier)

        return pred, likelihood
```

```
digit=['0', '1', '2', '3', '4', '5', '6','7','8','9']
naive_df = pd.DataFrame(list(zip(digit, digit_accuracy)), columns =['Digit','Digit Accuracy'])
print('Digit (Individual Class) Accuracy of the Samples:')
naive_df
```

```
Digit (Individual Class) Accuracy of the Samples:
```

	Digit	Digit Accuracy
0	0	0.908266
1	1	0.958444
2	2	0.754032
3	3	0.770833
4	4	0.687631
5	5	0.642015
6	6	0.924335
7	7	0.922857
8	8	0.747951
9	9	0.913793

Accuracy for digits 4, 5 are very less, as we see in the confusion matrix that the predictions of digit 5 are done for 6,4,7 instead of 5.
One probable reason is that we consider the probability of each class to be equal, but according to the dataset we don't have the equal number of classes for each.



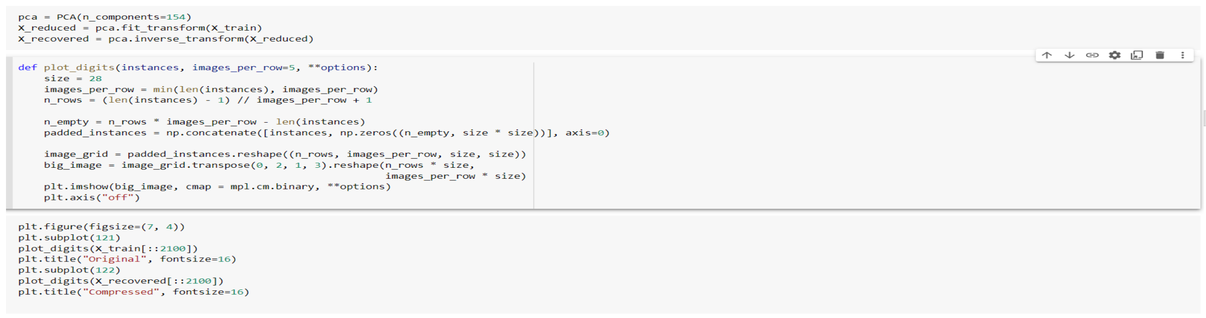
```
#Print Overall Accuracy
print('Overall Accuracy of Naive Bayes Model: '+str(overall_accuracy))
overall_accuracy
```

```
Overall Accuracy of Naive Bayes Model: 0.8069
0.8069
```

Accuracy: 80.70%: Naive bayes gives low accuracy as compared to kNN. The reason being, it considered all the probabilities as independent of each other (hence- naive).

We used PCA(Principal component analysis) for Compression, to reduce dimensionality reduction of the datasets.

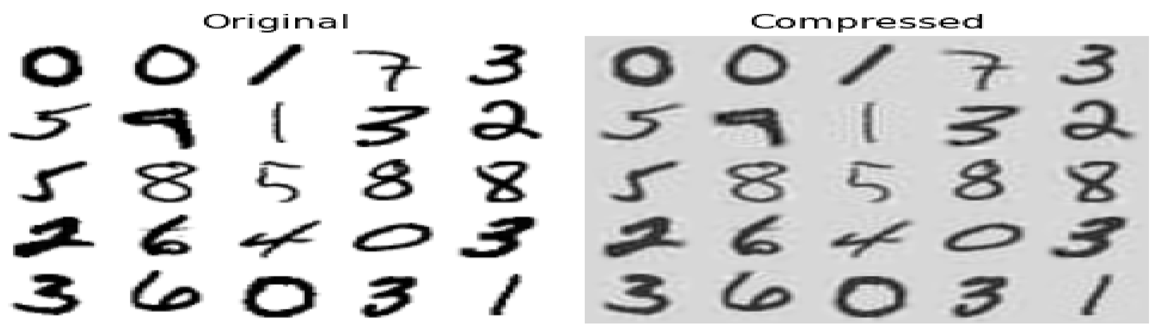
MNIST compression that preserves 95% of the variance:



```
pca = PCA(n_components=154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

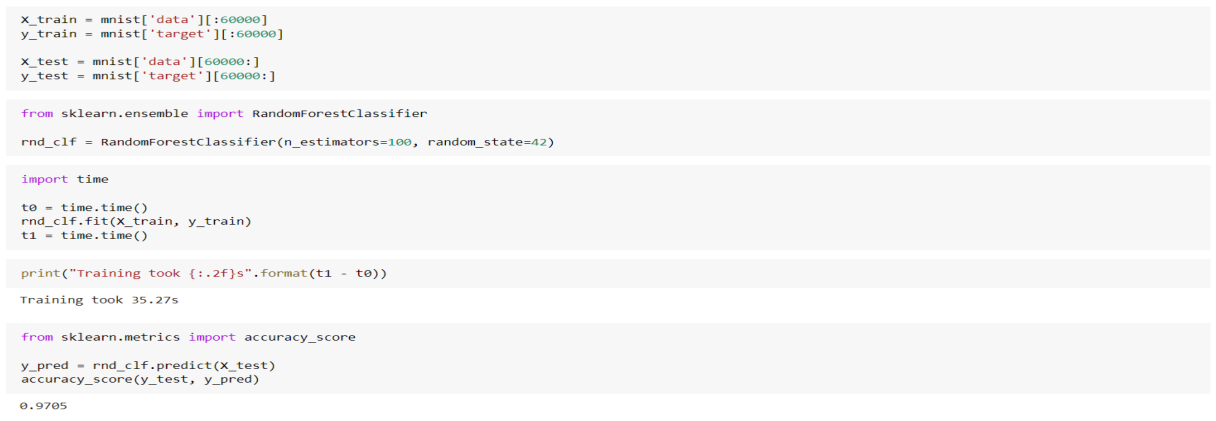
```
def plot_digits(instances, images_per_row=5, **options):
    size = 28
    images_per_row = min(len(instances), images_per_row)
    n_rows = (len(instances) - 1) // images_per_row + 1

    n_empty = n_rows * images_per_row - len(instances)
    padded_instances = np.concatenate([instances, np.zeros((n_empty, size * size))], axis=0)

    image_grid = padded_instances.reshape((n_rows, images_per_row, size, size))
    big_image = image_grid.transpose(0, 2, 1, 3).reshape(n_rows * size,
                                                         images_per_row * size)
    plt.imshow(big_image, cmap = mpl.cm.binary, **options)
    plt.axis("off")
```

```
plt.figure(figsize=(7, 4))
plt.subplot(121)
plot_digits(X_train[::2100])
plt.title("Original", fontsize=16)
plt.subplot(122)
plot_digits(X_recovered[::2100])
plt.title("Compressed", fontsize=16)
```



7th Classifier - Random Forest

Performance Evaluation:



```
X_train = mnist['data'][:60000]
y_train = mnist['target'][:60000]

X_test = mnist['data'][60000:]
y_test = mnist['target'][60000:]
```

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
import time

t0 = time.time()
rnd_clf.fit(X_train, y_train)
t1 = time.time()
```

```
print("Training took {:.2f}s".format(t1 - t0))
```

```
Training took 35.27s
```

```
from sklearn.metrics import accuracy_score

y_pred = rnd_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

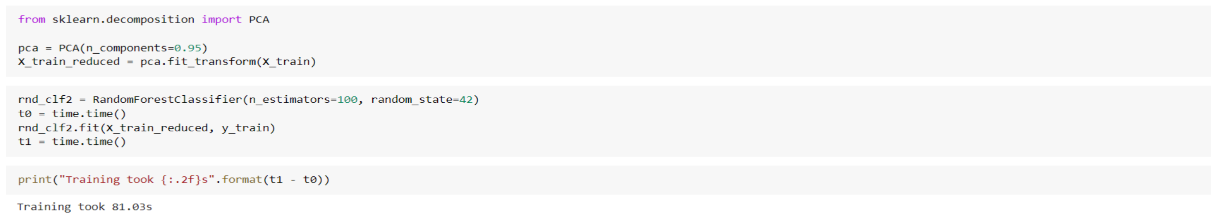
```
0.9705
```

Training Time : 35.27s

Accuracy : 97.05%

8th Classifier - Random Forest on Reduced PCA dataset (after reduction in dimensionality)

Performance Evaluation:



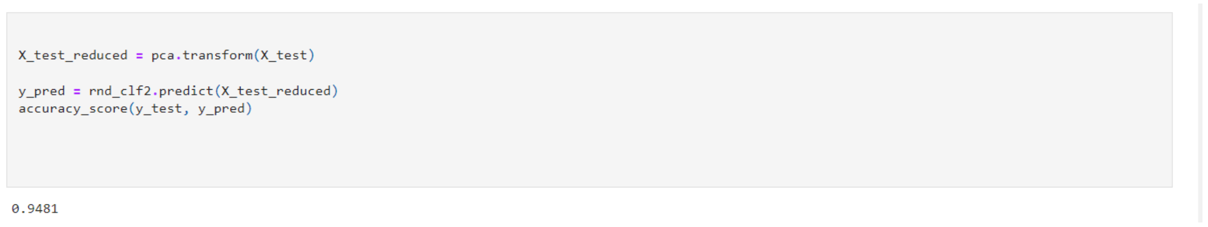
```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
X_train_reduced = pca.fit_transform(X_train)
```

```
rnd_clf2 = RandomForestClassifier(n_estimators=100, random_state=42)
t0 = time.time()
rnd_clf2.fit(X_train_reduced, y_train)
t1 = time.time()
```

```
print("Training took {:.2f}s".format(t1 - t0))
```

```
Training took 81.03s
```

Training Time : 81.03s **(Increased by 2.3 times).** After Dimensionality reduction it is taking more time to train.



```
X_test_reduced = pca.transform(X_test)

y_pred = rnd_clf2.predict(X_test_reduced)
accuracy_score(y_test, y_pred)
```

```
0.9481
```

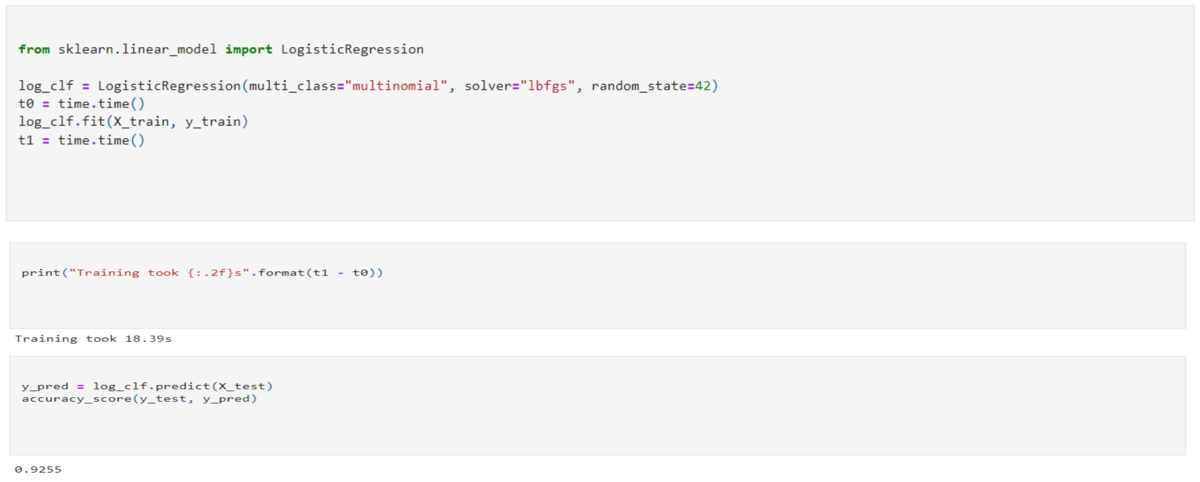
Accuracy: 94.81%

After dimensionality reduction there are various cases where there is slight reduction in the performance as we lose useful data signals in the process.

For this case PCA was not useful, as the training time was increased by 2.3 times and also performance was degraded significantly from 97.05% to 94.81%

9th Classifier - Logistic regression (Softmax)

Performance Evaluation:



```
from sklearn.linear_model import LogisticRegression

log_clf = LogisticRegression(multi_class="multinomial", solver="lbfgs", random_state=42)
t0 = time.time()
log_clf.fit(X_train, y_train)
t1 = time.time()
```

```
print("Training took {:.2f}s".format(t1 - t0))
```

```
Training took 18.39s
```

```
y_pred = log_clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

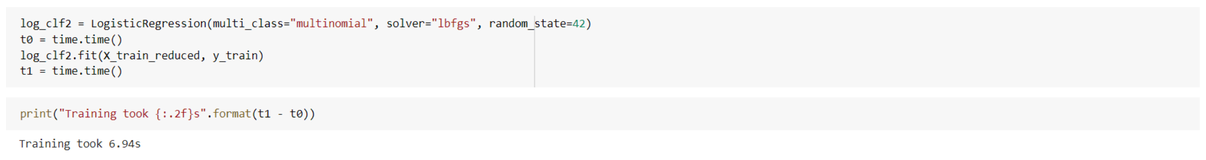
```
0.9255
```

Logistic Regression Training time: 18.39s

Accuracy: 92.55%

10th Classifier - Logistic regression on Reduced PCA dataset (after reduction in dimensionality)

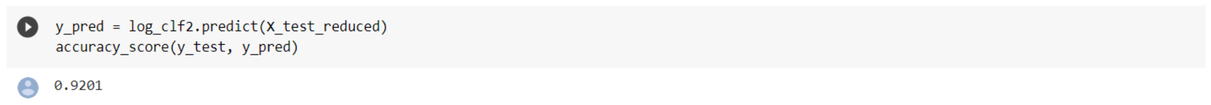
Performance Evaluation:



```
log_clf2 = LogisticRegression(multi_class="multinomial", solver="lbfgs", random_state=42)
t0 = time.time()
log_clf2.fit(X_train_reduced, y_train)
t1 = time.time()
```

```
print("Training took {:.2f}s".format(t1 - t0))
```

```
Training took 6.94s
```



```
y_pred = log_clf2.predict(X_test_reduced)
accuracy_score(y_test, y_pred)
```

```
0.9201
```

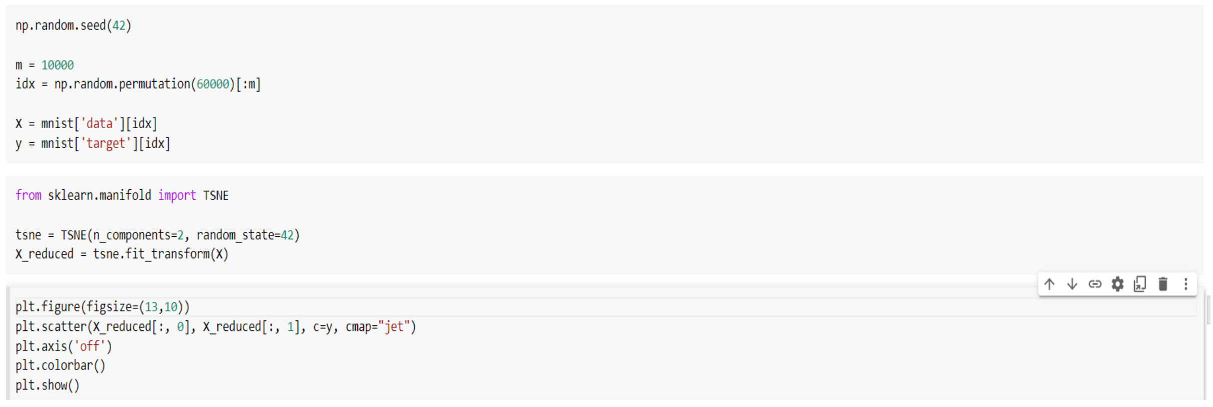
Logistic Regression Training time: 6.94s
Accuracy: 92.01%
Here we see that PCA has sped up the training time by approximately 2.5 times whereas performance after dimensionality reduction has slightly decreased from 92.55% to 92.01%.

Using PCA has given us a speedup with a very slight decrease in performance.

Thus, Doing Dimensionality reduction doesn't always give us speed. (In this case for Random forest classifier training time was worse whereas with Logistic regression was 2.5time better with almost same performance)

t-SNE (T- distributed Stochastic Neighbor Embedding):

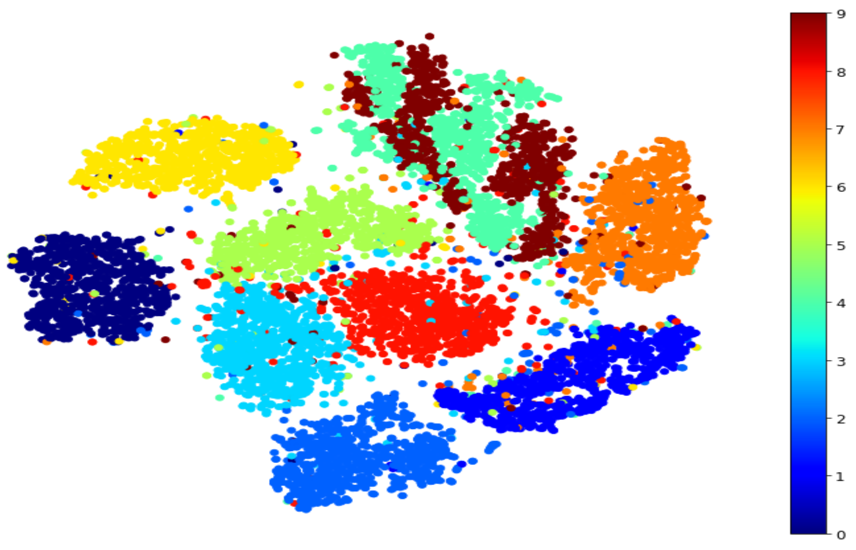
We used t-SNE to reduce the dataset dimensionality to 2D and have used scatter plot to plot using 10 different colours.
We have done for 10,000 random instances



```
np.random.seed(42)

m = 10000
idx = np.random.permutation(60000)[:m]

X = mnist['data'][idx]
y = mnist['target'][idx]
```

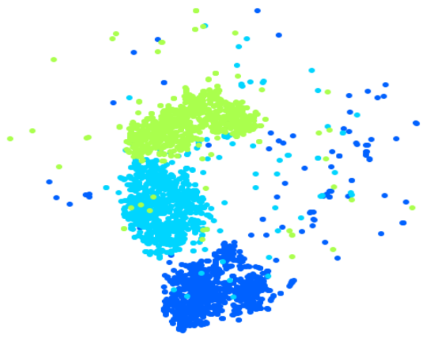
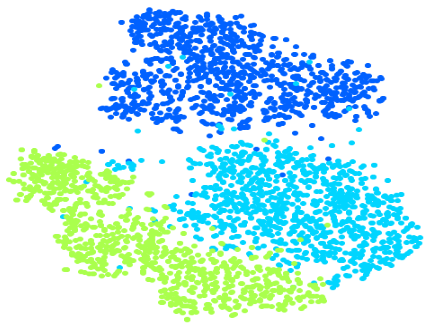
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, random_state=42)
X_reduced = tsne.fit_transform(X)
```

```
plt.figure(figsize=(13,10))
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=y, cmap="jet")
plt.axis('off')
plt.colorbar()
plt.show()
```

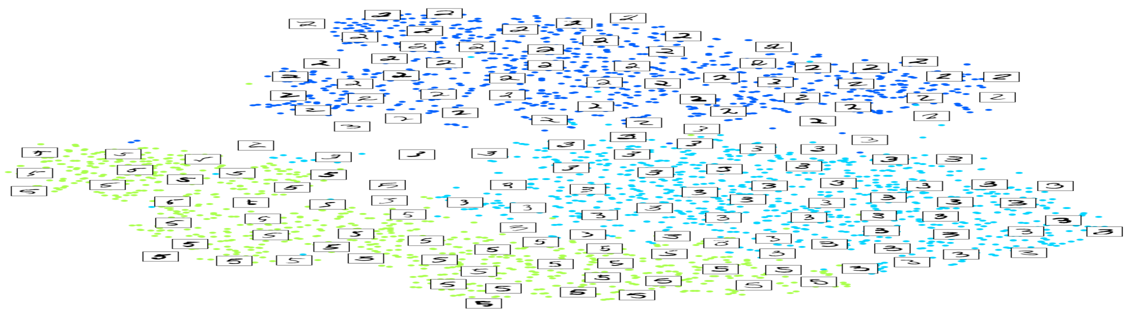


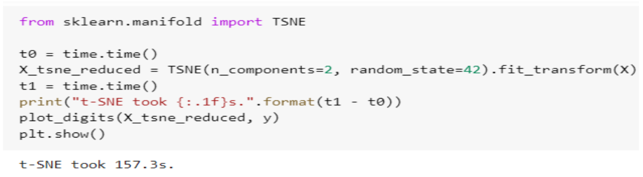
We can see in the plot how numbers are distinguished from each other.
Example we see 4, 9 to overlap, and 2,3, 5 also overlap, whereas 0, 6, 8 are easily distinguished from each other.
As 2,3, 5 overlap We had run t-SNE only on this 3 digits.

Before After

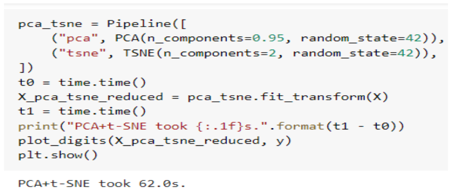
We see now the clusters are far and less overlapping. To exactly see what kind of image of a digit is overlapping, we have plotted the digits to visualize.





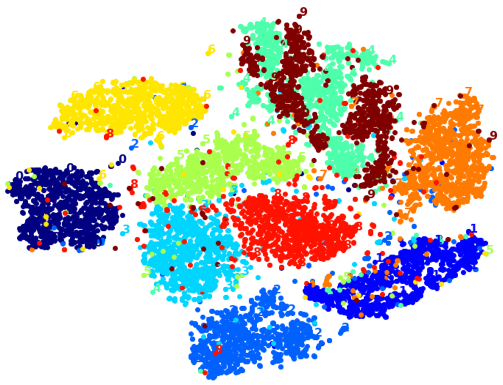
```
from sklearn.manifold import TSNE

t0 = time.time()
X_tsne_reduced = TSNE(n_components=2, random_state=42).fit_transform(X)
t1 = time.time()
print("t-SNE took {:.1f}s.".format(t1 - t0))
plot_digits(X_tsne_reduced, y)
plt.show()
```

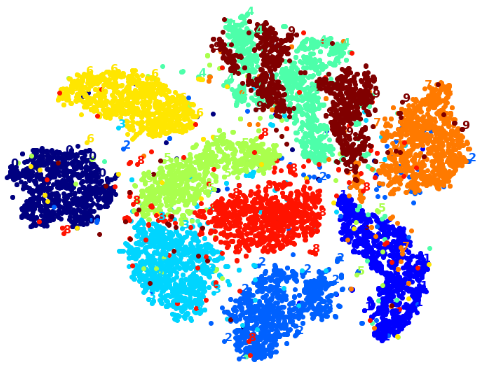
```
t-SNE took 157.3s.
```



```
pca_tsne = Pipeline([
    ("pca", PCA(n_components=0.95, random_state=42)),
    ("tsne", TSNE(n_components=2, random_state=42)),
])
t0 = time.time()
X_pca_tsne_reduced = pca_tsne.fit_transform(X)
t1 = time.time()
print("PCA+t-SNE took {:.1f}s.".format(t1 - t0))
plot_digits(X_pca_tsne_reduced, y)
plt.show()
```

```
PCA+t-SNE took 62.0s.
```





PCA here has increased the speed approximately by 2.5 times without much changing the result when used with tSNE.

The following are the primary drivers for dimensionality reduction:

- To speed up training algorithm
- To visualize the data
- To save space (compression)

Drawbacks:

- The performance is degraded because some information is lost.
- It can be computationally intensive.
- It adds some complexity to your Machine Learning pipelines.

11th Classifier- TensorFlow: Neural Networks with Keras

Training a deep MLP on the MNIST dataset by manually tuning the hyperparameters. By increasing the learning rate exponentially, plotting the loss, and identifying the point where the loss suddenly increases, we are essentially looking for the ideal learning rate.

```
K = tf.keras.backend

class ExponentialLearningRate(tf.keras.callbacks.Callback):
    def __init__(self, factor):
        self.factor = factor
        self.rates = []
        self.losses = []
    def on_batch_end(self, batch, logs):
        self.rates.append(K.get_value(self.model.optimizer.learning_rate))
        self.losses.append(logs["loss"])
        K.set_value(self.model.optimizer.learning_rate, self.model.optimizer.learning_rate * self.factor)
```

```
tf.keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

learning rate of 1e-3, and growing it by 0.5% for each iteration:

training

```
optimizer = tf.keras.optimizers.SGD(learning_rate=1e-3)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
expon_lr = ExponentialLearningRate(factor=1.005)
```

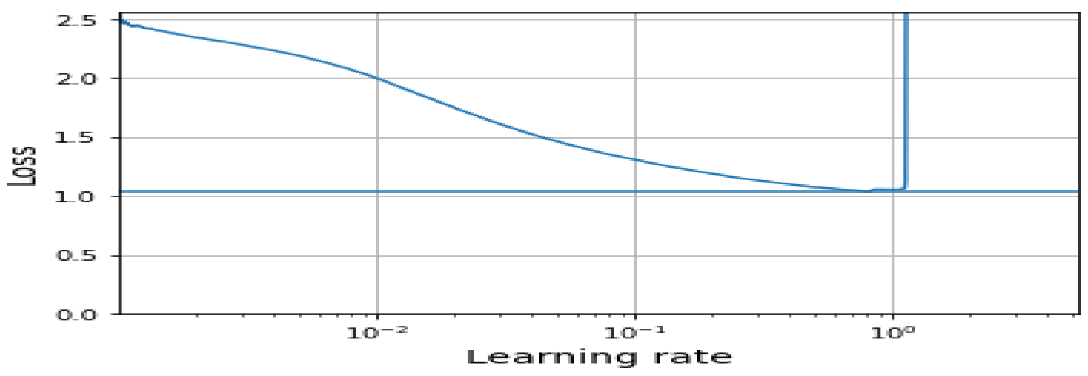
the model for just 1 epoch:

```
history = model.fit(X_train, y_train, epochs=1,
                    validation_data=(X_valid, y_valid),
                    callbacks=[expon_lr])
```

```
1719/1719 [==============================] - 3s 2ms/step - loss: nan - accuracy: 0.5843 - val_loss: nan - val_accuracy: 0.0958
```

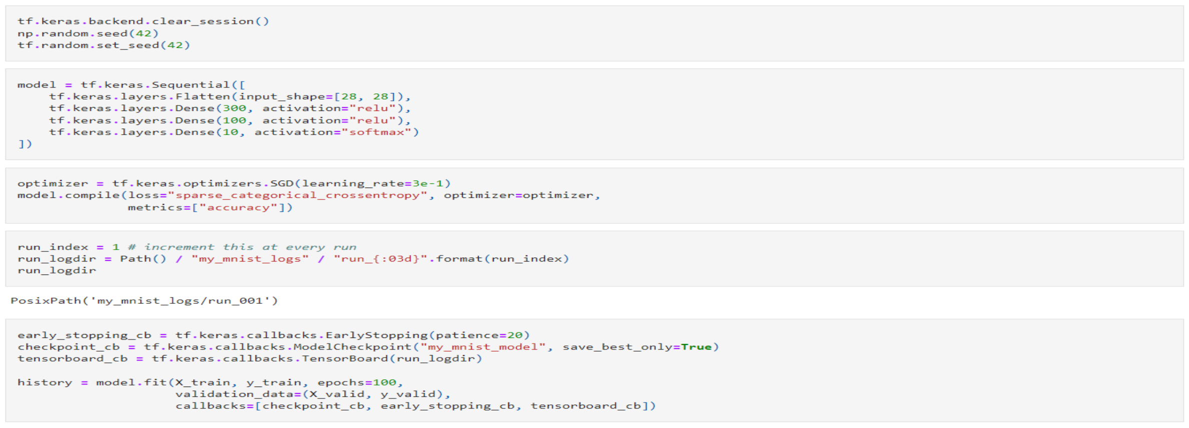
```
plt.plot(expon_lr.rates, expon_lr.losses)
plt.gca().set_xscale('log')
plt.hlines(min(expon_lr.losses), min(expon_lr.rates), max(expon_lr.rates))
plt.axis([min(expon_lr.rates), max(expon_lr.rates), 0, expon_lr.losses[0]])
plt.grid()
plt.xlabel("Learning rate")
plt.ylabel("Loss")
```

Loss as a function of the learning rate



From the graph we see, Loss is increasing suddenly to its peak when learning rate goes over 6e-1.

So using half of the learning rate at 3e-1 we trained and did performance evaluation of the classifier.



```
tf.keras.backend.clear_session()
np.random.seed(42)
tf.random.set_seed(42)
```

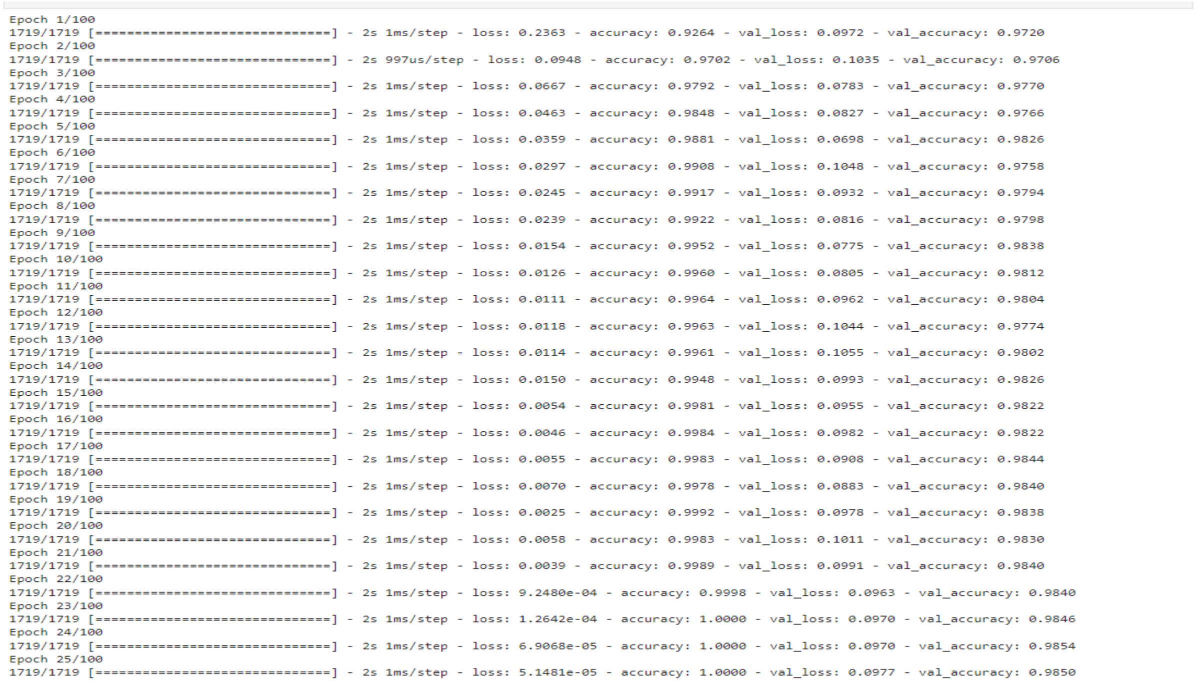
```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

```
optimizer = tf.keras.optimizers.SGD(learning_rate=3e-1)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
```

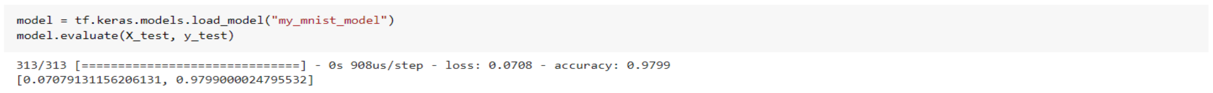
```
run_index = 1 # increment this at every run
run_logdir = Path() / "my_mnist_logs" / "run_{:03d}".format(run_index)
run_logdir
```

```
PosixPath('my_mnist_logs/run_001')
```

```
early_stopping_cb = tf.keras.callbacks.EarlyStopping(patience=20)
checkpoint_cb = tf.keras.callbacks.ModelCheckpoint("my_mnist_model", save_best_only=True)
tensorboard_cb = tf.keras.callbacks.TensorBoard(run_logdir)

history = model.fit(X_train, y_train, epochs=100,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, early_stopping_cb, tensorboard_cb])
```



```
Epoch 1/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.2363 - accuracy: 0.9264 - val_loss: 0.0972 - val_accuracy: 0.9720
Epoch 2/100
1719/1719 [==============================] - 2s 997us/step - loss: 0.0948 - accuracy: 0.9702 - val_loss: 0.1035 - val_accuracy: 0.9706
Epoch 3/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0667 - accuracy: 0.9792 - val_loss: 0.0783 - val_accuracy: 0.9770
Epoch 4/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0463 - accuracy: 0.9848 - val_loss: 0.0827 - val_accuracy: 0.9766
Epoch 5/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0359 - accuracy: 0.9881 - val_loss: 0.0698 - val_accuracy: 0.9826
Epoch 6/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0297 - accuracy: 0.9908 - val_loss: 0.1048 - val_accuracy: 0.9758
Epoch 7/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0245 - accuracy: 0.9917 - val_loss: 0.0932 - val_accuracy: 0.9794
Epoch 8/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0239 - accuracy: 0.9922 - val_loss: 0.0816 - val_accuracy: 0.9798
Epoch 9/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0154 - accuracy: 0.9952 - val_loss: 0.0775 - val_accuracy: 0.9838
Epoch 10/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0126 - accuracy: 0.9960 - val_loss: 0.0805 - val_accuracy: 0.9812
Epoch 11/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0111 - accuracy: 0.9964 - val_loss: 0.0962 - val_accuracy: 0.9804
Epoch 12/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0118 - accuracy: 0.9963 - val_loss: 0.1044 - val_accuracy: 0.9774
Epoch 13/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0114 - accuracy: 0.9961 - val_loss: 0.1055 - val_accuracy: 0.9802
Epoch 14/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0150 - accuracy: 0.9948 - val_loss: 0.0993 - val_accuracy: 0.9826
Epoch 15/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0054 - accuracy: 0.9981 - val_loss: 0.0955 - val_accuracy: 0.9822
Epoch 16/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0046 - accuracy: 0.9984 - val_loss: 0.0982 - val_accuracy: 0.9822
Epoch 17/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0055 - accuracy: 0.9983 - val_loss: 0.0908 - val_accuracy: 0.9844
Epoch 18/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0070 - accuracy: 0.9978 - val_loss: 0.0883 - val_accuracy: 0.9840
Epoch 19/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0025 - accuracy: 0.9992 - val_loss: 0.0978 - val_accuracy: 0.9838
Epoch 20/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0058 - accuracy: 0.9983 - val_loss: 0.1011 - val_accuracy: 0.9830
Epoch 21/100
1719/1719 [==============================] - 2s 1ms/step - loss: 0.0039 - accuracy: 0.9989 - val_loss: 0.0991 - val_accuracy: 0.9840
Epoch 22/100
1719/1719 [==============================] - 2s 1ms/step - loss: 9.2480e-04 - accuracy: 0.9998 - val_loss: 0.0963 - val_accuracy: 0.9840
Epoch 23/100
1719/1719 [==============================] - 2s 1ms/step - loss: 1.2642e-04 - accuracy: 1.0000 - val_loss: 0.0970 - val_accuracy: 0.9846
Epoch 24/100
1719/1719 [==============================] - 2s 1ms/step - loss: 6.9068e-05 - accuracy: 1.0000 - val_loss: 0.0970 - val_accuracy: 0.9854
Epoch 25/100
1719/1719 [==============================] - 2s 1ms/step - loss: 5.1481e-05 - accuracy: 1.0000 - val_loss: 0.0977 - val_accuracy: 0.9850
```



```
model = tf.keras.models.load_model("my_mnist_model")
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 0s 908us/step - loss: 0.0708 - accuracy: 0.9799
[0.07079131156206131, 0.9799000024795532]
```

Accuracy: 98%

Conclusion

The MNIST dataset provided us with an intriguing problem that is well confined and realistic. 11 Different classifiers and different dimensionality algorithms were analyzed. The SVM exhibited the lowest error rate among the classifiers, however this was at the expense of a pricey hyperparameter tuning procedure. Using TensorFlow, Neural Networks with Keras we got the highest accuracy of about 98%, KNN also performed almost as well with a very straightforward tuning process. The Ensemble classifiers didn't much boostup the performance. Although slightly behind this state of the art, the TensorFlow, KNN classifiers, and SVM (RBF) provided here nonetheless perform admirably given their individual computational complexity.

Doing Dimensionality reduction doesn't always give us the speed, and also in some rare cases it will give a little boost to the performance. But Most probably it basically speeds up the training with slight degrade in performance due to loss of some signals.

Naive Bayes Classifier classifies items based on probabilities by applying the Bayes method. KNN, on the other hand, makes the assumption that items that are similar occur nearby and are close to one another. So, it works by finding the maximum occurrence of a class around the given data set and classifies it to that one. We also observed that KNN outperforms Naive Bayes because the reason being, it considered all the probabilities as independent of each other (hence- naive).

Below are the figures for the performance of different classifiers and their training time on MNIST Dataset.

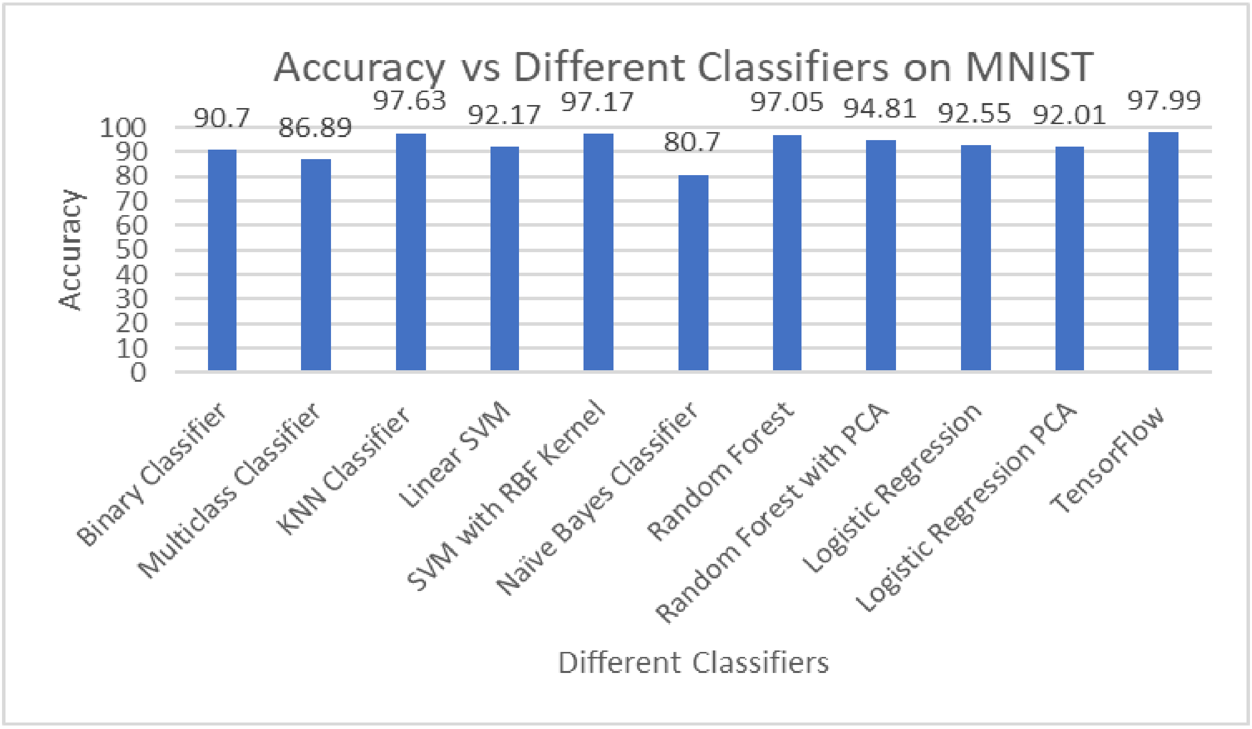


Figure 7: Accuracy of different classifiers on MNIST

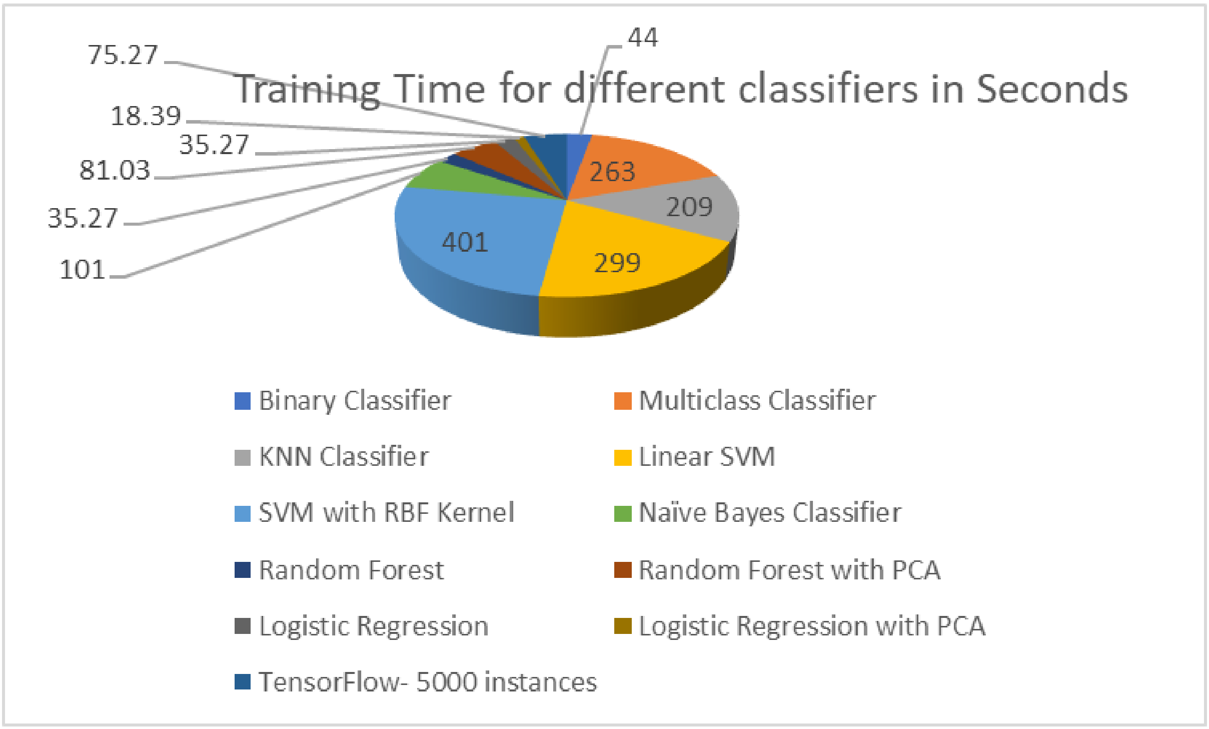


Figure 8: Training time for different classifiers in seconds for MNIST